Optic Algebras: Beyond In-memory Data Structures

J. Lopez-Gonzalez[a,b,1,*], J.M. Serrano[a,b]

[a] *Universidad Rey Juan Carlos*
[b] *Habla Computing*

Abstract

Abstract goes here!

1. Introduction

Optics provide abstractions and patterns to access and update immutable in-memory data structures. Specifically, they access and update *focus* values which are contextualized within a *whole* data structure. There are different kinds of optics in the folklore[2], each of them describing a particular type of transformation. Among them, *lens* [1] has become the most prominent one.

A lens do focus on a unique value that is always available from the whole data structure. They turn out to be very useful to deal with nested data structures [2]. As an example, consider the following entities, where *@Lenses* generates a lens for each field (or focus) in the annotated case class (or whole):

```
@Lenses case class Person(
  name: String,
  address: Address)

@Lenses case class Address(
  city: String,
  zip: Int)
```

Now suppose that we want to modify the zip code associated to a person. Since lenses composition is closed [3], we could express that transformation by combining the lenses associated to *address* and *zip* fields and then using the resulting composed lens to invoke the modifying method:

```
def modZip(f: Int => Int): Person => Person =
  (address compose zip).modify(f)
```

Despite the elegance that we get to describe this kind of transformations, lenses are restricted to work exclusively with in-memory data structures. Imagine that we wanted to persist this person in a relational database. In this scenario, we would discard optics in favor of the specific transformations provided by the new technology. Otherwise, we would need to pull the whole state, modify it by means of a lens and finally put it back again, which turns out to be completely impractical.

Firstly, we claim that it is possible to collect the essence of optics in a domain specific language [4], aimed to take optic abstractions and patterns beyond in-memory data structures. Secondly, we claim that this language is expressive enough to implement the data layer of an application and its associated logic once and for all, and later instantiate them to particular state-based interpretations: in-memory data structures, databases, microservices, etc. Lastly, we claim that we can do so while retaining reasonable levels of performance for a large class of applications. These are our contributions to fulfill our claims:

*Corresponding author
Email addresses: j.lopezgo@alumnos.urjc.es (J. Lopez-Gonzalez), j.serrano.hidalgo@urjc.es (J.M. Serrano)

[1] This is the specimen author footnote.

[2] http://oleg.fi/gists/images/optics-hierarchy.svg

- We propose *lens algebra* (section 2), which distills the algebraic essence of a lens. To better illustrate this idea, we need to see lenses from a coalgebraic perspective [5, 6, 7]. In this sense, optics are just machines —that can possibly run forever [8]— with buttons that can be pressed to observe or forward the machine state. In fact, what we mean by the algebraic essence is just the description of the machine interface in terms of its buttons, where any reference to the inner state has been abstracted away.
- Lens algebras in their *raw* formulation do not compose. In fact, it is difficult to determine what we mean by composition at this level of abstraction. To alleviate the situation, we propose a *natural* representation (section 3) —which heavily relies on natural transformations [9]— that enables composition for lens algebras.
- We take these ideas to other optics, particularly: *getter*, *setter*, *fold*, *affine* and *weak traversal*[3]. As a result, we end up with a collection of algebras that we refer to as *optic algebras* (section 4). These algebras represent analogous transformations to their in-memory optic counterparts, even preserving heterogeneous composition, but in a more general setting.
- We provide *Stateless* (section 5), an optic algebra library which is aimed to take these concepts and patterns to the development of industrial applications in the Scala ecosystem.

Previously, when we stated that optic algebras pursue transforming state in different settings, we actually meant that we want them to support different state-based effects [10]. In this sense, there exists work relating optics and effects, being *monadic lenses* [11] the most relevant one. Bidirectional transformations [12, 13] is also a great source of effectful transformations. We will compare our approach with these techniques in section 6.

2. Towards more abstract lenses

As stated in [1], lenses approach the view update problem for tree-structured data. They consist of a pair of functions, one of them to *get* the part from the whole and another one to *set* a new part into the current whole, returning a new version for the whole. We represent concrete lenses in Scala [2] as follows:

```
case class Lens[S, A](
  get: S => A,
  set: (S, A) => S)
```

As you may figured out, S and A role the whole and the part, respectively. Lens operations must obey certain laws to be considered very well-behaved [14]. Broadly, they keep the consistency between *get* and *set* and check that *set* does not modify anything else beyond the part. We show them below:

$$set(s,\ get(s)) = s \qquad \textbf{GetPut}$$
$$get(set(s,\ a)) = a \qquad \textbf{PutGet}$$
$$set(set(a1,\ s),\ a2) = set(s,\ a2) \qquad \textbf{PutPut}$$

In section 1 we inferred that lens generation for the fields of a case class is systematic, since we could generate them automatically. As an example, here it is the manual implementation of *zip*, which just determines where is the part that we may want to access or update:

```
val zip = Lens[Address, Int](
  s => s.zip,
  (s, a) => s.copy(zip = a))
```

[3]This is a relaxed traversal, where the transformation function for each focus must be exactly the same.

Now, it is time for us to distill lenses and extract their algebraic essence. This consists of abstracting the in-memory details away, in other words, removing any reference to the outer S state from the pair of functions *get* and *set*. To do so, we will try to represent both operations in terms of a state transformation $State[S,\ ?]$, where S is completely isolated, and therefore easier to abstract away.

State transformations take an initial state and additional input, to produce the new version of the state along with some output. What follows is an informal adaptation of the original *get* and *set* signatures into this state transformation template.

Remark. *Let s, a be types representing the whole and the focus of a lens, respectively. So,*

$$
\begin{aligned}
&(get, set) && \\
&= (S \Rightarrow A,\ (S,\ A) \Rightarrow S) && \textit{(a)} \\
&= (S \Rightarrow A,\ (A,\ S) \Rightarrow S) && \textit{(b)} \\
&= (S \Rightarrow A,\ A \Rightarrow S \Rightarrow S) && \textit{(c)} \\
&= (1 \Rightarrow S \Rightarrow A,\ A \Rightarrow S \Rightarrow S) && \textit{(d)} \\
&= (1 \Rightarrow S \Rightarrow A,\ A \Rightarrow S \Rightarrow (S,\ 1)) && \textit{(e)} \\
&= (1 \Rightarrow S \Rightarrow (S,\ A),\ A \Rightarrow S \Rightarrow (S,\ 1)) && \textit{(f*)} \\
&= (1 \Rightarrow State[S,\ A],\ A \Rightarrow State[S,\ 1]) && \textit{(g)} \\
&\therefore\ (1 \Rightarrow F[A],\ A \Rightarrow F[1]) && \textit{(h*)}
\end{aligned}
$$

So, we start from the original signatures of our *get* and *set* operations packed in a tuple (a). After swapping the arguments (b) and currying (c) from *set*, we provide an artificial input argument for *get* (d) typed 1. On the other hand, *set* is lacking the corresponding output, which is added in (e).

The next step consists of providing an artificial transformed output for *get* (f). We know that this method does not update the state, so this change of signature comes with a new law[4]: the initial state must be placed as the transformed one as is.

$$get_f(s)._1 \ = \ s \qquad\qquad \textbf{XGet}$$

Finally, once we have our signatures matching the state transformation template, we can describe them in terms of $State[S,\ ?]$, which is the only place where we reference S in both operations (g). Now, we can abstract away this reference to S by turning $State[S,\ ?]$ into a mere $F[?]$ (h). As you might noticed, this step does not come for free, it requires us to include new laws. We will come back to this aspect shortly. By now, we can see that the essence of a lens is precisely a lens algebra. We now package get_h and set_h in a *trait* to reify this concept:

```
trait LensAlg[F[_], A] {
  def get(): F[A]
  def set(a: A): F[Unit]
}
```

If we take into account that typeclasses [15] in Scala are modeled as traits [16], we can appreciate that this class is basically *MonadState* [17]. The major structural difference relies in the fact that our class does not extend $Monad$. To go further with this connection, we need to bring laws into the picture.

The final step of the derivation led us to get_j and set_j, that no longer returned plain types. Instead, their codomain is wrapped with a type constructor in both cases. Once we carry out this step, lens laws become broken. Take **GetSet** as an example, we cannot feed the codomain of get_j to set_j, since it is expecting an A typed argument.

[4]Steps marked with an asterisk represent that additional laws emerge.

If we want to recover lens laws, we are going to need additional combinators. These combinators are precisely the ones that we get when *LensAlg* extends *Monad*. By providing them, **GetSet** becomes $get_j >>= set_j = return()$, which is exactly one of the laws from *MonadState*. In fact, lens laws turn into *MonadState* laws when we abstract away from $State[S, ?]$. As a result, we determine that the essence of a lens is literally *MonadState*.

3. Natural implementation of lenses

Once our lenses algebras are defined, we could be interested in combining them, just as plain lenses are combined. However, when we move composition to this scenario, the situation becomes more complicated. As we saw in the person example (section 1), lenses compose when the focus of the first lens matches the whole of the second one. However, our optic algebras have a type constructor $F[?]$ as weird whole and a concrete type A as focus, which leads to an odd scenario.

Therefore, we need to ask ourselves what does it mean to compose a pair of lens algebras. To do so, we will suppose that we have defined the corresponding optic algebras to the optics we described in the person example:

```
def addressAlg: LensAlg[F[_], Address]
def zipAlg: LensAlg[G[_], Int]
```

At this point, we do not know much about them, but we may guess from the context that adressAlg should be hiding somehow the state of a *Person* while granting accessors to its *Address* part, while *zipAlg* encapsulates an *Address*, providing accessors to the *Int* zip code. Therefore, if we compose *addressAlg* with *zipAlg* we should end up with a lens whose type would be close to LensAlg[F[_], Address], i.e. an algebra that hides a person and put focus on the zip code associated to its address. However, this composition is not possible, since it requires a morphism G ~> F to be implemented.

Unfortunately, we know now that lens algebras do not compose. However, along the way, we found an interesting clue: we need some kind of morphism to compose them. This is interesting, since there are further connections between monad morphisms and lenses [12]. Having said so, we would like to instantiate lens algebra buttons with a morphism-alike representation, to restore composition. This morphism should transform programs on the part into programs on the whole. Following these intuitions, we get the following abstraction:

```
case class NatLensAlg[P[_], Q[_], A](
    hom: Q ~> P)(implicit
    MS: MonadState[A, Q]) extends LensAlg[P, A] {
  def get(): P[A] = hom(MS.get)
  def put(a: A): P[Unit] = hom(MS.put(a))
}
```

NatLensAlg takes three type parameters. The first of them represents the outer program, that knows how to access the whole. On the other hand, the second one, or inner program, knows how to access the focus. The type of the focus is determined by the third type parameter. This abstraction takes one value parameter hom, which corresponds with the monad homomorphism that enables composition.

However, if we want to instantiate LensAlg, we need to provide an implementation for get and set. There is no way to provide such implementations solely with the monad morphism hom. Thereby, we need to determine what particular inner programs are the ones that become get and set when passed through the homomorphism. If we assume that our inner program is an instance of MonadState[A, ?], this is the same as providing a homomorphism for that algebra among Q and P.

Now, we can declare a method to compose natural lens algebras. Indeed, it becomes trivial when using this representation, we simply compose monad morphisms:

```
def compose[P[_], Q[_], R[_], B](
    ln1: NatLensAlg[P, Q, A]
    ln2: NatLensAlg[Q, R, B])(implicit
```

```
    MS1: MonadState[A, Q],
    MS2: MonadState[B, R]): NatLensAlg[P, R, B] =
  NatLensAlg(ln1.hom compose ln2.hom)
```

Coming back to the guiding person example, we can reformulate our original lenses as natural lens algebras and compose them:

```
def addressNat: NatLensAlg[F[_], Address]
def zipNat: NatLensAlg[G[_], Int]
def compNat: NatLensAlg[F[_], Int] =
  addressNat compose zipNat
```

Notice that we are still working at a high level of abstraction. In fact, we have no idea about how are F[_] or G[_] handling their encapsulated state. As long as hom is a monad morphism, our composed lens expresses that we can access or update the nested part from the whole, no matter the particular configuration where it is deployed.

4. Generalizing to other optics

Lens composition is fundamental to deal with nested data structures, but optics really shine when they are composed heterogeneously [3]. Therefore, in order to take optic benefits to the algebraic setting, we need to describe other optic algebras and be able to compose them. In this section, we introduce *affine algebras* and *traversal algebras*. For each of them, we will show a very brief introduction, we will extract its algebraic essence and finally, we will show a composable representation.

4.1. Affine Algebra

A lens implies that the part is always present in the whole, however this could not be the case in other scenarios. Affine[5] comes to alleviate this situation by evidencing that the part could be unavailable from the whole:

```
case class Affine[S, A](
  getOption: S => Option[A],
  set: (S, A) => S)
```

If we derive the essence of the affine, as we did in section 2 for lens, we end up with the following definition for affine algebras:

```
trait AffineAlg[F[_], A] extends Monad[F] {
  def getOption(): F[Option[A]]
  def set(a: A): F[Unit]
}
```

As for the lens algebra case, it does not make much sense to compose affine algebras. We need to find a morphism representation to enable this feature. Intuitively, an inner program that modifies the part should be transformed into an outer program that optionally modifies the whole, just in case the part does exist. This intuition lead us to the following representation:

```
case class NatAffineAlg[P[_], Q[_], A](
    hom: Q ~> P[Option[?]])(implicit
    MS: MonadState[A, Q]) extends AffineAlg[P, A] {
  def getOption(): P[Option[A]] = hom(MS.get)
  def put(a: A): P[Option[Unit]] = hom(MS.put(a))
}
```

[5] This optic is also known as *affine traversal*, *affine lens* or even *optional* in the folklore.

In fact, if we instantiate this algebra for NatAffineAlg[State[S, ?], State[A, ?], A] we get something isomorphic to Affine[S, A], which supports the new algebra definition. Now that we have a composable representation, we should be able to compose a pair of affine algebras:

```
def compose[P[_], Q[_], R[_], B](
    af1: NatAffineAlg[P, Q, A]
    af2: NatAffineAlg[Q, R, B])(implicit
    MS1: MonadState[A, Q],
    MS2: MonadState[B, R]): NatAffineAlg[P, R, B] =
  NatAffineAlg((af1.hom compose af2.hom).map(_.join))
```

And what is even more interesting, we should be able to compose optics heterogeneously, for instance, a lens algebra with an affine algebra:

```
def compose[P[_], Q[_], R[_], B](
    ln: NatLensAlg[P, Q, A]
    af: NatAffineAlg[Q, R, B])(implicit
    MS1: MonadState[A, Q],
    MS2: MonadState[B, R]): NatAffineAlg[P, R, B] =
  NatLensAlg(ln.hom compose af.hom)
```

This composition results in an affine algebra, which is pretty consistent with classic optic composition, where the result of composing a lens with an affine is itself an affine. Notice as well that reversing the composing optic types, eg: composing an affine algebra with a lens algebra, does not affect the resulting type.

4.2. (Weak) Traversal Algebra

While lenses and affines do focus on a single part, which could or could not exist, we lack something to focus on a sequence of parts, which is exactly what traversals are for. This optic turns out to be tricky and it is not possible to represent it as a bunch of simple functions [7]. Therefore, we will use a relaxed version, which we call *Weak Traversal*:

```
case class WTraversal[S, A](
  getAll: S => List[A],
  modify: (A => A) => S => S)
```

This optic is constrained in the sense that the modification that we achieve over each part must be exactly the same, but we find it good enough for most of cases. If we derive the essence of a weak traversal, we get the following interface:

```
trait WTraversalAlg[F[_], A] extends Monad[F] {
  def getList(): F[List[A]]
  def modify(f: A => A): F[List[Unit]]
}
```

As usual, we need to find a composable representation for these traversal algebras. If we apply the same ideas that we used for affine algebras, we get the next representation, where the possibility of failing focus is replaced with the possibility of multiple focus:

```
case class NatWTraversalAlg[P[_], Q[_], A](
    hom: Q ~> P[List[?]])(implicit
    MS: MonadState[A, Q]) extends WTraversalAlg[P, A] {
  def getList() = hom(MS.get)
  def modify(f: A => A) = hom(MS.put(a))
}
```

Again, if we instantiate this algebra for NatWTraversalAlg[State[S, ?], State[A, ?], A] we get something isomorphic to WTraversal[S, A]. Now that we have a composable representation, we should be able to compose homogeneously:

```
def compose[P[_], Q[_], R[_], B](
    af1: NatWTraversalAlg[P, Q, A]
    af2: NatWTraversalAlg[Q, R, B])(implicit
    MS1: MonadState[A, Q],
    MS2: MonadState[B, R]): NatWTraversalAlg[P, R, B] =
  NatWTraversalAlg((af1.hom compose af2.hom).map(_.join))
```

And besides, we should be able to compose optics heterogeneously, for instance, a lens algebra with a traversal algebra:

```
def compose[P[_], Q[_], R[_], B](
    ln: NatLensAlg[P, Q, A]
    af: NatTraversalAlg[Q, R, B])(implicit
    MS1: MonadState[A, Q],
    MS2: MonadState[B, R]): NatTraversalAlg[P, R, B] =
  NatTraversalAlg(ln.hom compose af.hom)
```

As expected by optic composition rules, composing lens algebras with traversal algebras results in traversal algebras. Had we composed with an affine algebra instead, it would have resulted in a traversal algebra.

5. Stateless: an optic algebra Scala library

Once we have introduced the fundamentals of optic algebras, it is time to put them into practice. In fact, we have implemented Stateless, a Scala library where we deploy algebras and utilities to deal with them. We will use a guiding example to introduce the library basics. Particularly, we will extend the *zip code* one. Now, we will consider that there exists a department, with a budget and a list of members. For each member, we will be able to access its name and optionally, to access its address. Each address, in turn, will contain the associated city and zip code. Our objective is to implement modifyZip, that receives a function to modify a zip code and returns a program that modifies all the zip codes belonging to the people in the department according to the input function.

Our example contains three major entities: address, person and department. Each of them will be located in its own data structure. We will start with address, which contains the name of the city and the zip code itself. Since they appear exactly once in every address, we know that accessing those fields could be achieved by means of lenses. Therefore we represent addresses as follow:

```
trait Address[Ad] {
  type P[_]
  val city: LensAlg[P, String]
  val zip: LensAlg[P, Int]
}
```

First of all, this trait takes a type parameter that represents the final state of the address, or the context to get access to it. Secondly, it contains a **type** member which roles programs that are able to access and modify the address state. Finally, you can find a lens algebras for each field. Concretely, this is a natural lens algebra. Contrary to what we saw in section 3, Stateless use the same name for both raw and natural optic algebras (which are kept in different packages), and hides the Q type parameter as a type member for the second case, to improve compiler inference while composing optics. Now, we show how to encode a person:

```
trait Person[Pr] {
  type P[_]
  type Ad
  val name: LensAlg[P, String]
  val address: Address[Ad]
  val optAddress: OptionalAlg.Aux[P, address.P, Ad]
}
```

As before, this entity takes a type parameter Dp, which represents the final state, and a type member P, which corresponds with the program that transforms the state. Besides, this entity contains two fields and therefore they are mapped into two optic algebras. There is nothing remarkable about name, but optAddress brings new patterns. Specifically, its returning type contemplates Q, since we need to connect its inner program with the outer program from Address, to make them composable. As a consequence, we need to provide an evidence of entity address, and therefore the type of its final state Ad. The last entity, department, is represented as folows:

```
trait Department[Dp] {
  type P[_]
  type Pr
  val budget: LensAlg[P, Long]
  val person: Person[Pr]
  val people: TraversalAlg.Aux[P, person.P, Pr]
}
```

It contains two optic algebras, a simple lens for budget and a dependent traversal for people. Concretely, it has to establish the dependency between department and person. To do so, we use the same pattern we described above for optAddress.

The resulting data layer does not differ greatly from representing the state of the application with several case classes. In this sense, a simple field is represented with a lens algebra, an optional field is represented with an affine algebra and a list of elements is represented with a traversal algebra.

The most striking feature of this data layer is that we can describe modifyZip once and for all, in a modular and elegant way:

```
def modifyZip(f: Int ⇒ Int): P[Unit] =
  (people compose optAddress compose zip).modify(f)
```

As you see, Stateless contains infix methods to compose a particular optic with another passed as argument, following the classic optic hierarchy. In this regard, it also contains abstracting methods, to turn an optic algebra into a hierarchy ancestor. Finally, it also provides the means to transform an optic algebra into its corresponding indexed one.

Notice that modifyZip knows nothing about the particular programs or the final application state. In fact, they could be mere *State* programs dealing with case classes as state, or they could be *IO* programs using a database session as state. We show these particular interpretations in the appendix. There you will find the utilities that Stateless provides to facilitate the instantiating task.

6. Related Work

7. Conclusion

Acknowledgements

MINECO

References

[1] J Nathan Foster, Michael B Greenwald, Jonathan T Moore, Benjamin C Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. *ACM SIGPLAN Notices*, 40(1):233–246, 2005.

[2] Tony Morris. Asymmetric lenses in scala. 2012.

[3] Matthew Pickering, Jeremy Gibbons, and Nicolas Wu. Profunctor optics: Modular data accessors. *arXiv preprint arXiv:1703.10857*, 2017.

[4] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196, 1996.

[5] Bart P.F. Jacobs. Objects and classes, coalgebraically. Technical report, Amsterdam, The Netherlands, The Netherlands, 1995.

[6] Bart Jacobs and Jan Rutten. A tutorial on (co) algebras and (co) induction. *Bulletin-European Association for Theoretical Computer Science*, 62:222–259, 1997.
[7] Russell O'Connor. Functor is to lens as applicative is to biplate: Introducing multiplate. *arXiv preprint arXiv:1103.2841*, 2011.
[8] David A Turner. Total functional programming. *J. UCS*, 10(7):751–768, 2004.
[9] Benjamin C Pierce. *Basic category theory for computer scientists*. MIT press, 1991.
[10] Philip Wadler. Monads for functional programming. In *International School on Advanced Functional Programming*, pages 24–52. Springer, 1995.
[11] Faris Abou-Saleh, James Cheney, Jeremy Gibbons, James McKinna, and Perdita Stevens. Reflections on monadic lenses. *CoRR*, abs/1601.02484, 2016.
[12] Faris Abou-Saleh, James Cheney, Jeremy Gibbons, James McKinna, and Perdita Stevens. Notions of bidirectional computation and entangled state monads. In *International Conference on Mathematics of Program Construction*, pages 187–214. Springer, 2015.
[13] Faris Abou-Saleh and Jeremy Gibbons. Coalgebraic aspects of bidirectional computation. *Journal of Object Technology*, 16(1), 2017.
[14] Sebastian Fischer, Zhenjiang Hu, and Hugo Pacheco. A clear picture of lens laws. In *International Conference on Mathematics of Program Construction*, pages 215–223. Springer, 2015.
[15] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76. ACM, 1989.
[16] Bruno CdS Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *ACM Sigplan Notices*, volume 45, pages 341–360. ACM, 2010.
[17] Jeremy Gibbons. Unifying theories of programming with monads. In *International Symposium on Unifying Theories of Programming*, pages 23–67. Springer, 2012.